neo4j

The World's Leading Graph Database

Ebook

The Definitive Guide to Graph Databases

for the RDBMS Developer

by Michael Hunger, Ryan Boyd & William Lyon

TABLE OF CONTENTS

For those cases when you need a different solution, we hope this book helps you recognize when – and how – to use a graph database to tackle those new challenges.

The Definitive Guide to Graph Databases *for the RDBMS Developer*

Michael Hunger, Ryan Boyd & William Lyon

Introduction:

Why We Wrote This Ebook

When two database technologies share space in the same book title, there's bound to be confusion as to the motives of its writing.

First things first: We didn't write this book to bash relational databases or to criticize a still-valuable technology. Without relational databases, many of today's most mission-critical applications wouldn't run, and without the early innovation of RDBMS pioneers, we would have never gotten to where we are with today's database technology.

Rather, we wrote this book to introduce you – a developer with RDBMS experience – to a database technology that changed not only how we see the world, but how we build the future. Today's business and user requirements demand applications that connect more and more of the world's data, yet still expect high-levels of performance and data reliability.

We believe those applications of the future will be built using graph databases, and we don't want you to be left behind. In fact, we're here to help you through every step of the learning process.

While other NoSQL (Not only SQL) databases advertise themselves with in-your-face defiance of RDBMS technology, we prefer to be a helpful resource in helping you add graph databases to your professional skillset.

Relational databases still have their perfect use cases. But for those cases when you need a different solution, we hope this book helps you recognize when – and how – to use a graph database to tackle those new challenges.

As you read and peruse these pages, feel free to reach out to us with your questions. You can most commonly find us on Stack Overflow, our Google Group or our public Slack channel.

Happy graphing,

–Michael, Ryan & Will

Chapter 1:

Why Relational Databases Aren't Always Enough

Relational databases are powerful tools.

Since the 80s, they have been the power-horse of most software applications and continue to be so today. Relational databases (RDBMSs) were initially designed to codify paper forms and tabular structures, and they do that exceedingly well. For the right use case and the right architecture, they are one of the best tools for storing and organizing data.

Because relational databases store highly structured data in tables with predetermined columns and many rows of the same type of information, they require developers and applications to strictly structure the data used in their applications.

But today's user requirements and applications are asking for ***more***. More features, more data, more agility, more speed and – most importantly – more connections.

The Mismatch between Relational Databases & Data Relationships

Despite their name, relational databases are not well-suited for today's highly connected data, because they don't robustly store relationships ***between*** data elements.

(It's worth noting that relational databases take their name from the highly specific mathematical notion of a "relation" – a.k.a. a table – as part of E.F. Codd's relational algebra. The name does not derive from describing relationships between data.)

Traditionally, developers have been taught to store data in the columns and rows of a relational model. Yet, columns and rows aren't really how data exists in the real world. Rather, data exists as objects and the relationships between those different objects.

These types of complex, real-world data are increasing in volume, velocity and variety. As a result, data relationships – which are often more valuable than the data itself – are growing at an even faster rate.

The problem: Relational databases aren't designed to capture this rich relationship information.

The bottom line: Applications (and the enterprises that create them) are missing out on critical connections essential for today's high-stakes, data-driven decisions.

The Agile Realities of Today's Software Applications

Every development team faces the reality of ever-changing business and user requirements that call for frequent modifications and pivots to a given data architecture.

Database administrators (DBAs) and developers face a steady stream of business requests to add elements or attributes to meet new requirements – such as storing information about the latest social platform – but such regular schema changes are problematic for RDBMS developers and come with a high maintenance cost.

That's because relational databases don't adapt well to change. Rather, their fixed schema works best for problems that are well-defined at the outset.

Despite their name, relational databases are not well-suited for today's highly connected data, because they don't robustly store relationships between data elements.

Applications (and the enterprises that create them) are missing out on critical connections essential for today's high-stakes, data-driven decisions.

Slow and expensive schema redesigns also hurt the agile software development process by hindering your team's ability to innovate quickly – a significant opportunity cost no matter the size of your bottom line.

The verdict: Relational databases aren't engineered for the speed of business agility.

How Connected Data Queries Cripple RDBMS Performance

Despite advances in computing, faster processors and high-speed networks, the performance of some relational database applications continues to slow. This performance slump has several known symptoms (see "SQL Strain" section below), but the root cause usually boils down to one factor: queries about data relationships.

Since relational databases aren't built or optimized to handle connected data, any attempt to answer data relationship queries – such as a recommendation engine, a fraud detection pattern or a social graph – involves numerous JOINs between database tables.

In relational databases, references to other rows and tables are indicated by referring to their primary-key attributes via foreign-key columns. (See Figure 1 below for an example.)

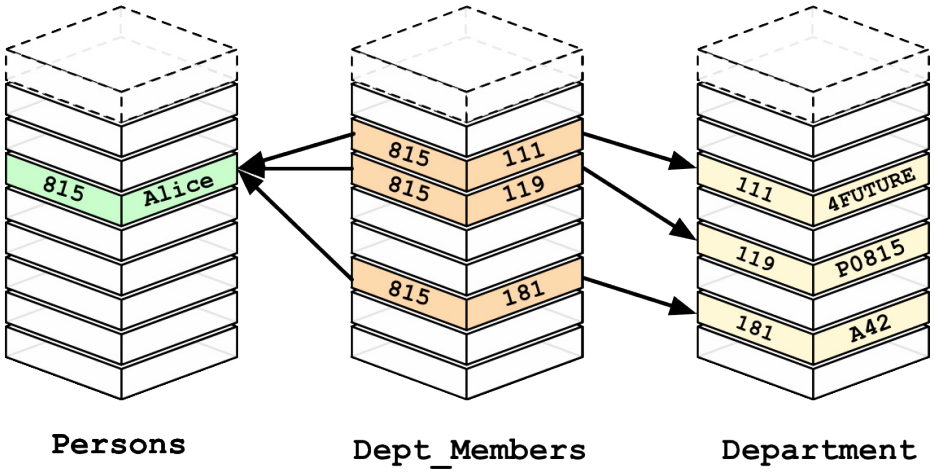

Figure 1: A JOIN table between the `Persons` and `Departments` tables in a relational database using foreign key constraints.

These references are enforceable with constraints, but only when the reference is never optional. JOINs are then computed at query time by matching primary- and foreign-keys of the many (potentially indexed) rows of the to-be-JOINed tables. These operations are compute- and memory-intensive and have an exponential cost as queries grow.

Consequently, modeling and storing connected data becomes impossible without extreme complexity. That complexity surfaces in cases like SQL statements that require dozens of lines of code just to accomplish simple operations. Overall performance also degrades from query complexity, the number and levels of data relationships and the overall size of the database.

With today's real-time, always-on expectations of software applications, traditional relational databases are simply inappropriate whenever data relationships are key to success.

5 Signs Your RDBMS Application Suffers from SQL Strain

Many relational database applications are working fine within their limits. Some, however, may be showing significant signs of strain induced by the database, especially when an RDBMS is being used to handle highly connected data.

Here are five of the most common signs you may be trying to solve a connected data problem with a relational database:

1. A Large Number of JOINs

When you utilize queries that JOIN many different tables, there's an explosion of complexity and computing resource consumption. This results in a corresponding increase in query response times.

2. Numerous Self-JOINs (or Recursive JOINs)

Self-JOIN statements are common for hierarchy and tree representations of data, but traversing relationships by repeatedly JOINing tables to themselves is inefficient. In fact, some of the longest SQL queries in the world involve recursive JOINs.

3. Frequent Schema Changes

At a time when business agility is at a premium, requests for changes are more often than not put off by DBAs because the schema of relational databases isn't designed for frequent modifications and pivots. Common schema changes indicate that the data or requirements are rapidly evolving, calling for a more flexible data model.

4. Slow-Running Queries (Despite Extensive Tuning & Hardware)

Your DBA might use every trick in the book to speed up query times, but many SQL queries still aren't fast enough to support your application's needs. In addition, denormalizing data models for performance can negatively impact data quality and update behavior.

Or in some cases, you might have a handle on query performance only because of excessive hardware. Throwing more hardware at the problem might temporarily fix a problem, but queries shouldn't require over 100 cores in order to perform well. As your data grows, even more hardware will be required.

5. Pre-Computing Your Results

Because queries run so slowly, many applications pre-compute their results using a snapshot of the past data. However, this is effectively using yesterday's data for queries that should be handled in real time today. Furthermore, your system usually must pre-compute 100% of your data, even if only 1-2% of it will be accessed at any given time, wasting your computational resources.

An Alternative (or Addition) to Relational Databases

As previously mentioned, relational databases have their appropriate use cases. For highly structured, predetermined schemas, an RDBMS is the perfect tool.

But as we've seen, relational databases aren't always enough. Applications that require connected data insights can't rely on the relational model.

While relational databases ***sometimes*** need to be replaced entirely, often the RDBMS solution can't (or doesn't need to) be shut down. In these cases, developers and architects can use a polyglot persistence approach – using different databases for their best-of-breed strengths.

So whether you're replacing your RDBMS or just complementing it with another data store, the volume, velocity and variety of today's data – and data relationships – require a solution that's engineered from the ground up to store and organize connected data.

It's time to meet graph databases.

Chapter 2:

Why Graph Databases?

We already know that relational databases aren't enough (by themselves) for handling the volume, velocity and variety of today's data, but what's the clear alternative?

There are a lot of other database options out there – including a number of NoSQL data stores – but none of them are explicitly designed to handle and store data relationships. None, that is, except graph databases.

The biggest value that graphs bring to the development stack is their ability to store relationships and connections as first-class entities.

For instance, the early adopters of graph technology reimagined their businesses around the value of data relationships. These companies have now become industry leaders: LinkedIn, Google, Facebook and PayPal.

As pioneers in graph technology, each of these enterprises had to build their own graph database from scratch. Fortunately for today's developers, that's no longer the case, as graph database technology is now available off the shelf.

Let's take a further look into why you should consider a graph database for your next connected-data application. We'll start with some basic definitions.

If you're already familiar with relational databases, you'll find graphs to be a breeze.

What Is a Graph?

You don't need to understand the arcane mathematical wizardry of graph theory in order to understand graph databases. On the contrary, if you're already familiar with relational databases, you'll find graphs to be a breeze.

First thing: A ***graph*** – in mathematics – is not the same as a chart, so don't picture a bar or line chart.

Rather, picture a network or mind map, like in the example to the right.

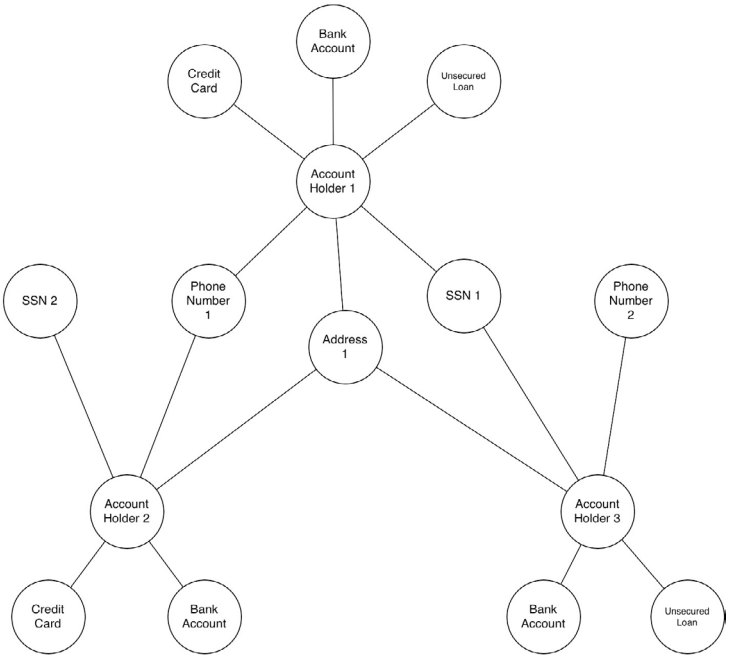

Figure 2: A basic graph of a fraud ring sharing similar contact information.

A graph is composed of two elements: a node and a relationship.

Each node represents an entity (a person, place, thing, category or other piece of data), and each relationship represents how two nodes are associated. For example, the two nodes "cake" and "dessert" would have the relationship "is a type of" pointing from "cake" to "dessert."

This general-purpose structure allows you to model all kinds of scenarios – from a system of roads, to a network of devices, to a population's medical history or anything else defined by relationships.

What Is a Graph Database?

A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

Unlike other databases, relationships take first priority in graph databases. This means your application doesn't have to infer data connections using foreign keys or out-of-band processing, such as MapReduce.

Unlike other databases, relationships take first priority in graph databases. This means your application doesn't have to infer data connections using foreign keys or out-of-band processing, such as MapReduce.

By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models that map closely to our problem domain.

There are two important properties of graph database technologies:

Graph Storage

Some graph databases use native graph storage that is specifically designed to store and manage graphs, while others use relational or object-oriented databases instead. Non-native storage is often much more latent, especially as data volume and query complexity grow.

Graph Processing Engine

Native graph processing (a.k.a. "index-free adjacency") is the most efficient means of processing graph data because connected nodes physically "point" to each other in the database. Non-native graph processing uses other means to process CRUD operations that aren't optimized for graphs, often involving an index lookup which results in reduced performance.

What Are the Advantages of Using a Graph Database?

A graph database is purpose-built to handle highly connected data, and the increase in the volume and connectedness of today's data presents a tremendous opportunity for sustainable competitive advantage.

When it comes to applying a graph database to a real-world problem, with real-world technical and business constraints, enterprise organizations choose graph databases for the following reasons:

Minutes-to-Milliseconds Performance

Query performance and responsiveness are at the top of many organizations' concerns with regard to their data platforms. Online transactional systems – large web applications in particular – must respond to end users in milliseconds if they are to be successful. In the relational world, as an application's dataset size grows, JOIN pains begin to manifest themselves, and performance deteriorates. Using index-free adjacency, a graph database turns complex JOINs into fast graph traversals – which are constant time operations – thereby maintaining millisecond performance irrespective of the overall size of the dataset.

Drastically Accelerated Development Cycles

The graph data model reduces the impedance mismatch that has plagued software development for decades, thereby reducing the development overhead of translating back and forth between an object model and a tabular relational model.

More importantly, the graph model reduces the impedance mismatch between the technical and business domains. Subject matter experts, architects and developers can talk about and picture the core domain using a shared model that is then incorporated into the application itself.

Extreme Business Responsiveness

Successful applications rarely stay still. Changes in business conditions, user behaviors, and technical and operational infrastructures drive new requirements. In the past, this has required organizations to undertake careful and lengthy data migrations that involve modifying schemas, transforming data and maintaining redundant data to serve old and new features.

Developing with graph databases aligns perfectly with today's agile, test-driven development practices, allowing your graph database to evolve in step with the rest of the application and any changing business requirements. Rather than exhaustively modeling a domain ahead of time, data teams can add to the existing graph structure without endangering current functionality.

Enterprise Ready

When employed in a mission-critical application, a data technology must be robust, scalable and – more often than not – transactional. Although some graph databases are fairly new and not yet fully mature, there are graph databases on the market that provide all the ***-ilities*** needed by large enterprises today:

- ACID transactionality
- High availability
- Horizontal read scalability
- Storage of billions of entities

These characteristics have been an important factor leading to the adoption of graph databases by large organizations, not merely in modest offline or departmental capacities, but in ways that truly transform the business.

What Are the Common Use Cases of Graph Databases?

While graph databases first became popular with social applications for the consumer web (Facebook, LinkedIn, Twitter), their use cases extend far beyond the social space.

Today's enterprise organizations use graph database technology in a diversity of ways, including these six most common use cases:

- Fraud detection
- Real-time recommendation engines
- Master data management (MDM)
- Network and IT operations
- Identity and access management (IAM)
- Graph-based search

For more information on graph technology use cases, see *The Top 5 Use Cases of Graph Databases: Unlocking New Possibilities with Connected Data*.

Chapter 3:

Data Modeling: Relational vs. Graph Models

In some regards, graph databases are like the next generation of relational databases, but with first class support for "relationships," or those implicit connections indicated via foreign keys in traditional relational databases.

Each node (entity or attribute) in the graph database model directly and physically contains a list of relationship-records that represent its relationships to other nodes. These relationship records are organized by type and direction and may hold additional attributes.

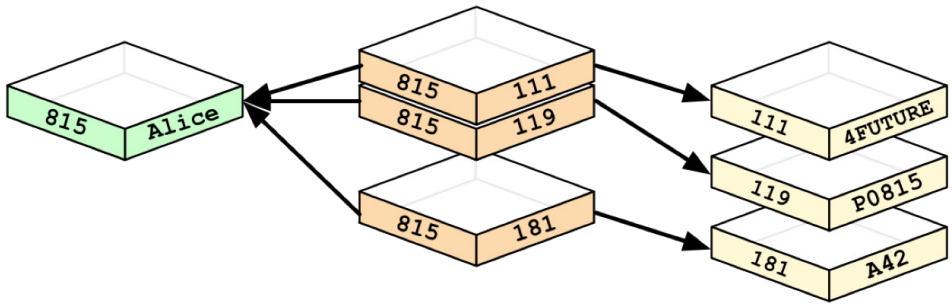

Figure 3: A graph/JOIN table hybrid showing the foreign key data relationships between the `Persons` and `Departments` tables in a relational database.

Graph databases like Neo4j provide a minutes-to-milliseconds performance advantage of several orders of magnitude, especially for JOIN-heavy queries.

Whenever you run the equivalent of a JOIN operation, the database just uses this list and has direct access to the connected nodes, eliminating the need for an expensive search-and-match computation.

This ability to pre-materialize relationships into database structures allows graph databases like Neo4j to provide a minutes-to-milliseconds performance advantage of several orders of magnitude, especially for JOIN-heavy queries.

The resulting data models are much simpler and at the same time more expressive than those produced using traditional relational or other NoSQL databases.

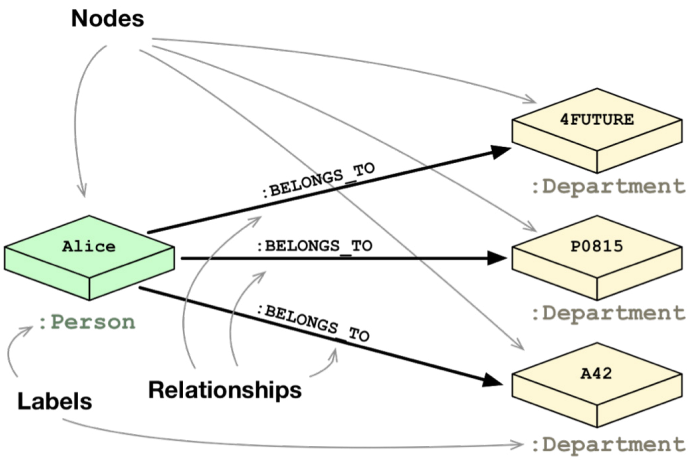

Figure 4: A graph data model of our original `Persons` and `Departments` data. Nodes and relationships have replaced our tables, foreign keys and JOIN table.

Graph databases support a very flexible and fine-grained data model that allows you to model and manage rich domains in an easy and intuitive way.

You more or less keep the data as it is in the real world: small, normalized, yet richly connected entities. This allows you to query and view your data from any imaginable point of interest, supporting many different use cases (see Chapter 2 for more information).

The fine-grained model also means that there is no fixed boundary around aggregates, so the scope of update operations is provided by the application during the read or write operation. Transactions group a set of node and relationship updates into an Atomic, Consistent, Isolated and Durable (ACID) operation.

Graph databases like Neo4j fully support these transactional concepts, including write-ahead logs and recovery after abnormal termination, so you never lose your data that has been committed to the database.

If you're experienced in modeling with relational databases, think of the ease and beauty of a well-done, normalized entity-relationship diagram: a simple, easy-to-understand model you can quickly whiteboard with your colleagues and domain experts. A graph is exactly that: a clear model of the domain, focused on the use cases you want to efficiently support.

Let's take a model of the organizational domain and show how it would be modeled in a relational database vs. the graph database.

First up, our relational database model:

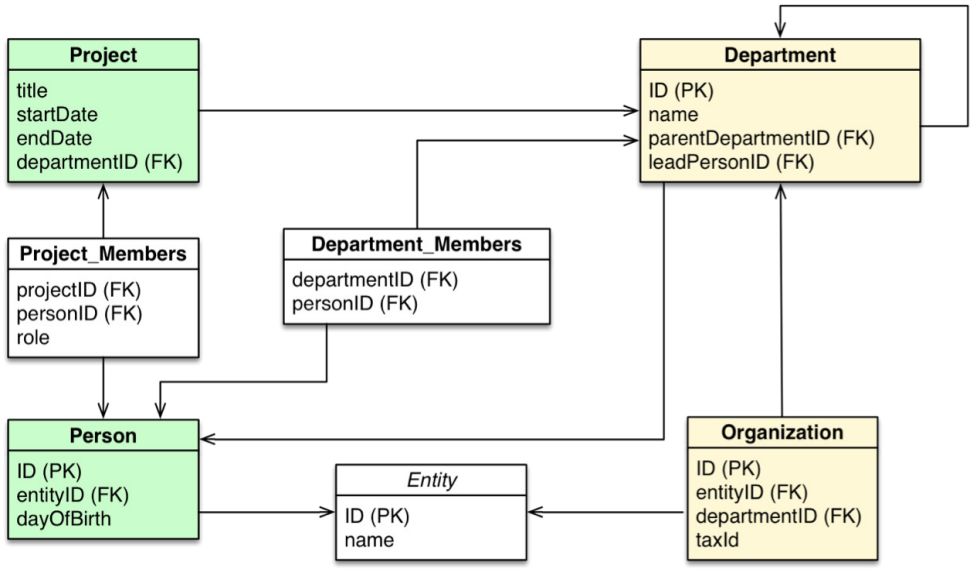

Figure 5: A relational database model of a domain with `Persons` and `Projects` within an `Organization` with several `Departments`.

Graph databases support a very flexible and fine-grained data model that allows you to model and manage rich domains in an easy and intuitive way.

If we were to adapt this (above) relational database model into a graph database model, we would go through the following checklist to help with the transformation:

- Each entity table is represented by a label on nodes
- Each row in a entity table is a node
- Columns on those tables become node properties.
- Remove technical primary keys, but keep business primary keys
- Add unique constraints for business primary keys, and add indexes for frequent lookup attributes
- Replace foreign keys with relationships to the other table, remove them afterwards
- Remove data with default values, no need to store those
- Data in tables that is denormalized and duplicated might have to be pulled out into separate nodes to get a cleaner model
- Indexed column names might indicate an array property (like `email1`, `email2`, `email3`)
- JOIN tables are transformed into relationships, and columns on those tables become relationship properties

Once we've taken these steps to simplify our relational database model, here's what the graph data model would look like:

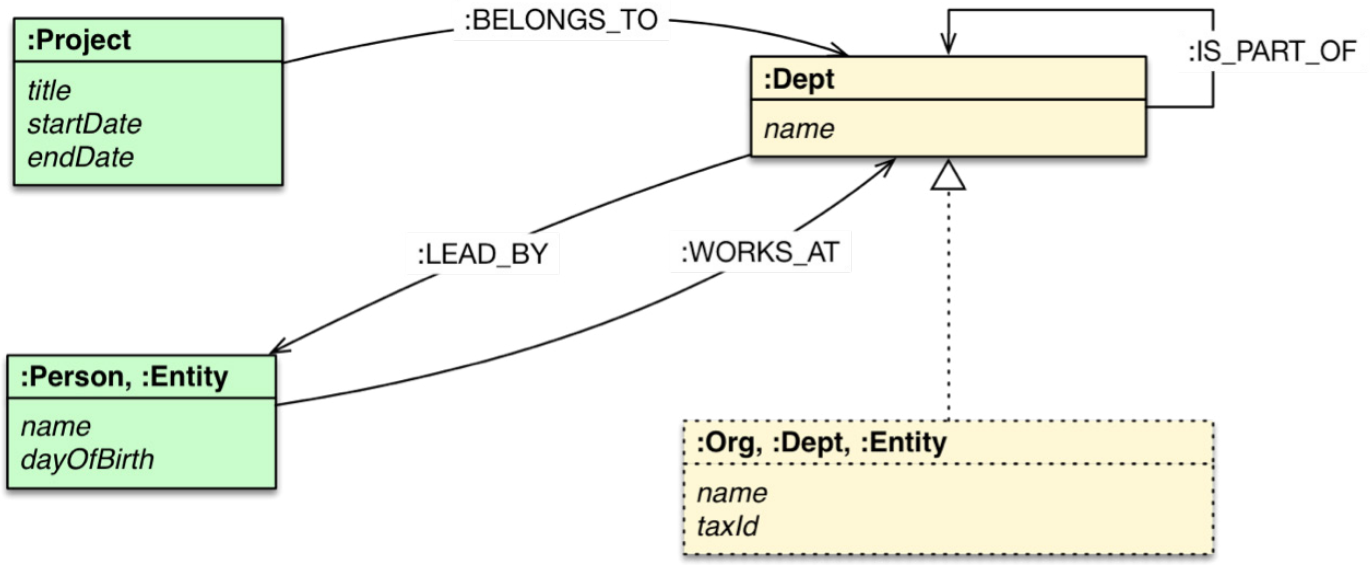

Figure 6: A graph data model of the same domain with `Persons` and `Projects` within an `Organization` with several `Departments`. With the graph model, all of the initial JOIN tables have now become data relationships.

This above example is just one simplified comparison of a relational and graph data model. Now it's time to dive deeper into a more extended example taken from a real-world use case.

Relational vs. Graph Data Modeling Case Study: A Data Center Management Domain

To show you the true power of graph data modeling, we're going to look at how we model a domain using both relational- and graph-based techniques. You're probably already familiar with RDBMS data modeling techniques, so this comparison will highlight a few similarities – and many differences.

In particular, we'll uncover how easy it is to move from a conceptual graph model to a physical graph model, and how little the graph model distorts what we're trying to represent versus the relational model.

To facilitate this comparison, we'll examine a simple data center management domain. In this domain, several data centers support many applications on behalf of many customers using different pieces of infrastructure, from virtual machines to physical load balancers.

A graph is a clear model of the domain, focused on the use cases you want to efficiently support.

Here's an example of a small data center domain:

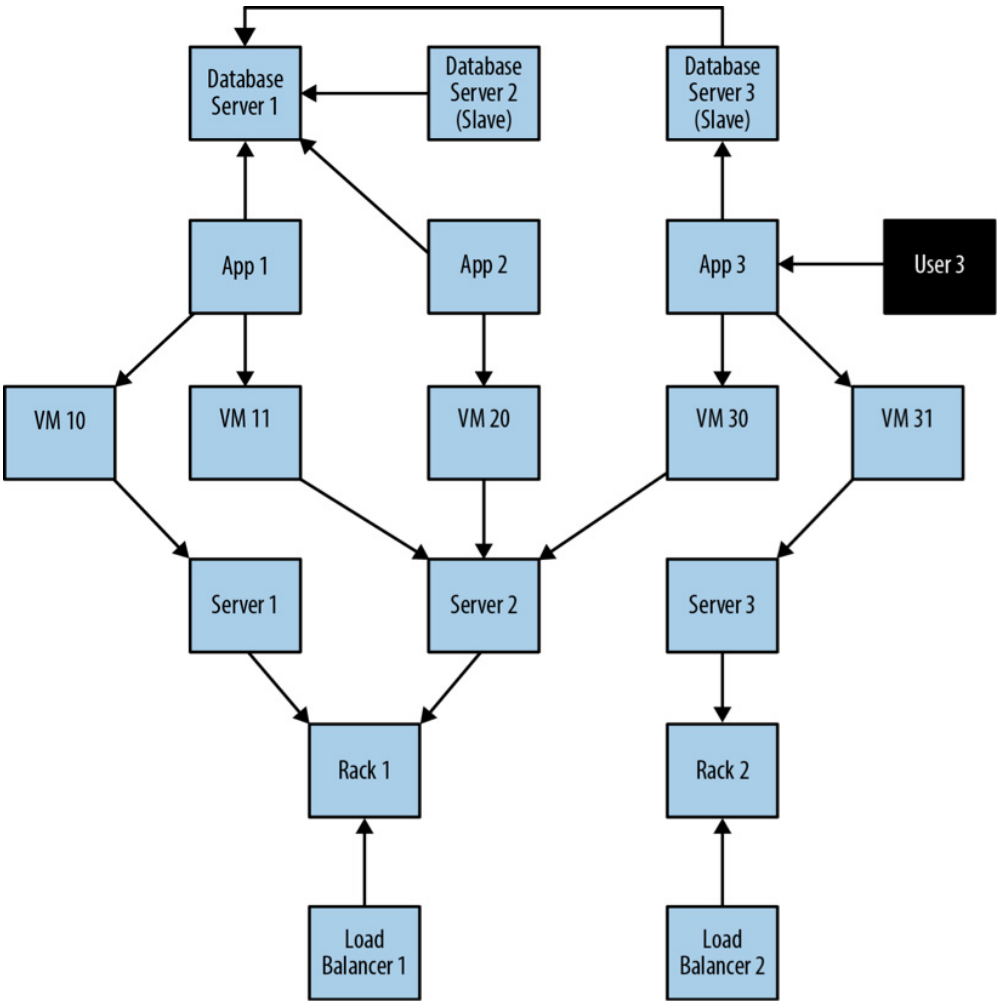

Figure 7: A small domain of several application deployments within a data center.

In this example above, we see a somewhat simplified view of several applications and the data center infrastructure necessary to support them. The applications, represented by nodes `App 1`, `App 2` and `App 3`, depend on a cluster of databases labeled `Database Server 1`, 2, 3.

While users logically depend on the availability of an application and its data, there is additional physical infrastructure between the users and the application; this infrastructure includes virtual machines (`Virtual Machine 10`, 11, 20, 30, 31), real servers (`Server 1`, 2, 3), racks for the servers (`Rack 1`, 2) and load balancers (`Load Balancer 1`, 2), which front the apps.

Of course, between each of the components are many networking elements: cables, switches, patch panels, NICs (network interface controllers), power supplies, air conditioning and so on – all of which can fail at inconvenient times. To complete the picture we have a straw-man single user of `Application 3`, represented by `User 3`.

As the operators of such a data center domain, we have two primary concerns:

- Ongoing provision of functionality to meet (or exceed) a service-level agreement, including the ability to perform forward-looking analyses to determine single points of failure, and retrospective analyses to rapidly determine the cause of any customer complaints regarding the availability of service.
- Billing for resources consumed, including the cost of hardware, virtualization, network provisioning and even the costs of software development and operations (since these are simply logical extensions of the system we see here).

If we are building a data center management solution, we'll want to ensure that the underlying data model allows us to store and query data in a way that efficiently addresses these primary concerns. We'll also want to be able to update the underlying model as the application portfolio changes, the physical layout of the data center evolves and virtual machine instances migrate.

Given these needs and constraints, let's see how the relational and graph models compare.

Creating the Relational Model

The first step in relational data modeling is the same as with any other data modeling approach: to understand and agree on the entities in the domain, how they interrelate and the rules that govern their state transitions.

The first step in relational data modeling is the same as any other data modeling approach: to understand and agree on the entities in the domain.

This initial stage is often informal, with plenty of whiteboard sketches and discussions between subject matter experts and data architects. These talks then usually result in diagrams like Figure 7 above (which also happens to be a graph).

The next step is to convert this initial whiteboard sketch into a more rigorous entity-relationship (E-R) diagram (which is ***another*** graph). Transforming the conceptual model into a logical model using a stricter notation gives us a second chance to refine our domain vocabulary so that it can be shared with relational database specialists.

(It's worth noting that adept RDBMS developers often skip directly to table design and normalization without using an intermediate E-R diagram.)

Here's our sample E-R diagram:

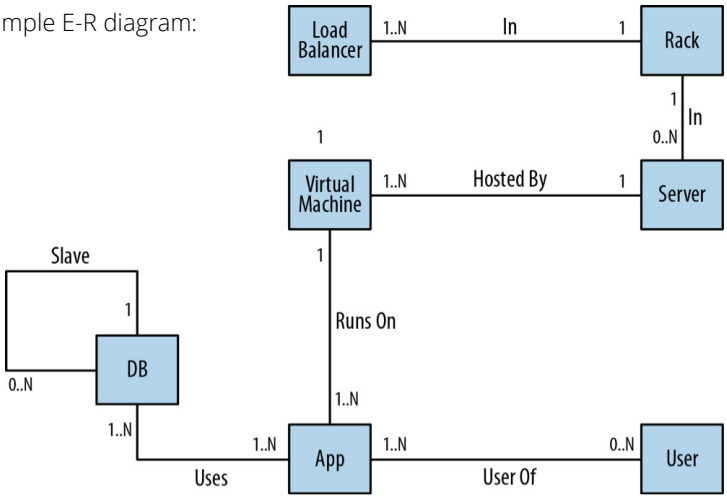

Figure 8: An entity-relationship (E-R) diagram for our data center domain.

A Note on E-R Diagrams: Despite being graphs, E-R diagrams immediately show the shortcomings of the relational model. E-R diagrams allow only single, undirected relationships between entities. In this respect, the relational model is a poor fit for real-world domains where relationships between entities are both numerous and semantically rich.

Now with a logical model complete, it's time to map it into tables and relations, which are normalized to eliminate data redundancy. In many cases, this step can be as simple as transcribing the E-R diagram into a tabular form and then loading those tables via SQL commands into the database.

But even the simplest case serves to highlight the idiosyncrasies of the relational model. For example, in the figure below we see that a great deal of accidental complexity has crept into the model in the form of foreign key constraints (everything annotated [FK]), which support one-to-many relationships, and JOIN tables (e.g., `AppDatabase`), which support many-to-many relationships – and all this before we've added a single row of real user data.

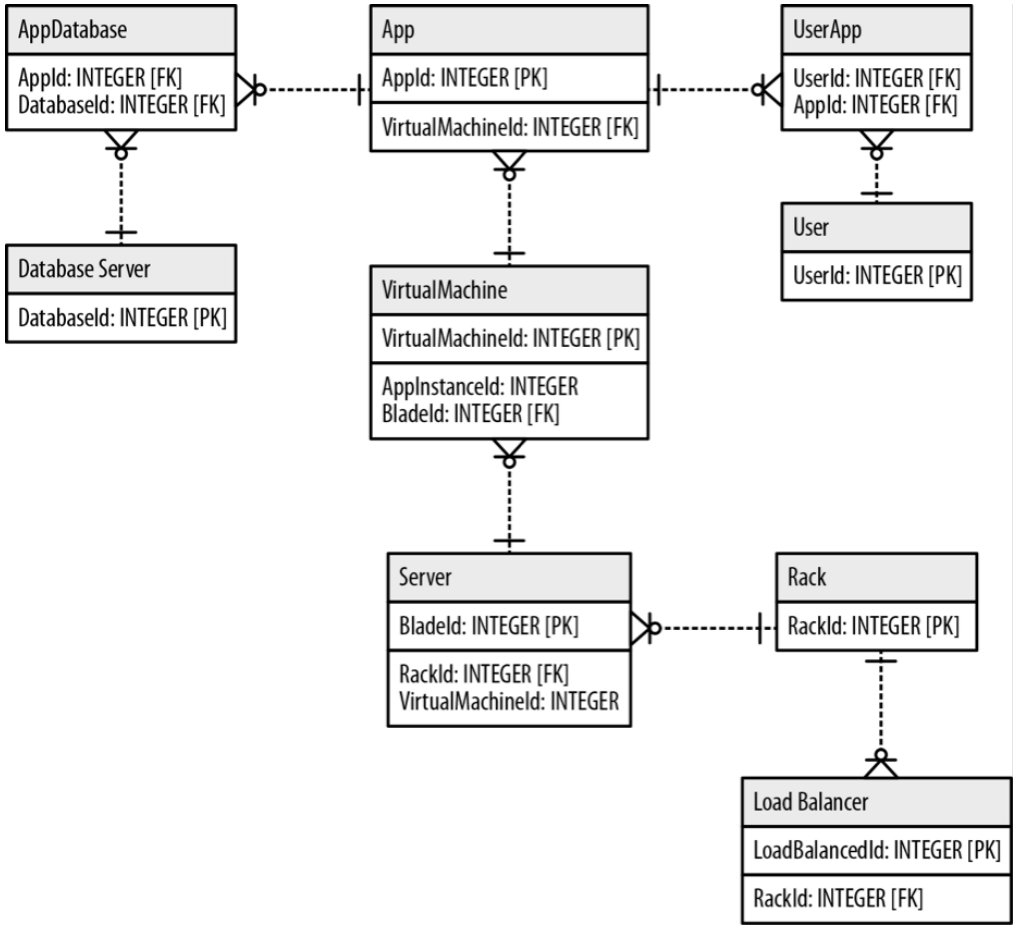

Figure 9: A full-fledged relational data model for our data center domain.

These constraints and complexities are model-level metadata that exist simply so that we specify the relations between tables at query time. Yet the presence of this structural data is keenly felt, because it clutters and obscures the domain data with data that serves the database, *not the user*.

The Problem of Relational Data Model Denormalization

So far, we now have a normalized relational data model that is relatively faithful to the domain, but our design work is not yet complete.

One of the challenges of the relational paradigm is that normalized models generally aren't fast enough for real-world needs. In theory, a normalized schema is fit for answering any kind of ad hoc query we pose to the domain, but in practice, the model must be further adapted for specific access patterns.

In other words, to make relational databases perform well enough for regular application needs, we have to abandon any vestiges of true domain affinity and accept that we have to change the user's data model to suit the database engine, not the user. This approach is called denormalization.

Denormalization involves duplicating data (substantially in some cases) in order to gain query performance.

For example, consider a batch of users and their contact details. A typical user often has several email addresses, which we would then usually store in a separate `EMAIL` table. However, to reduce the performance penalty of JOINing two tables, it's quite common to add one or more columns within the `USER` table to store a user's most important email addresses.

Assuming every developer on the project understands the denormalized data model and how it maps to their domain-centric code (which is a big assumption), denormalization is not a trivial task.

Often, development teams turn to an RDBMS expert to munge a normalized model into a denormalized one that aligns with the characteristics of the underlying RDBMS and physical storage tier. Doing all of this involves a substantial amount of data redundancy.

The Cost of Rapid Change in the Relational Model

It's easy to think the design-normalize-denormalize process is acceptable because it's only a one-off task. After all, the cost of this upfront work pays off across the lifetime of the system, right? Wrong.

While this one-off, upfront idea is appealing, it doesn't match the reality of today's agile development process. Systems change frequently – not only during development, but also during their production lifetimes.

Although the majority of systems spend most of their time in production environments, these environments are rarely stable. Business requirements change and regulatory requirements evolve, so our data models must too.

Adapting our relational database model then requires a structural change known as a migration. Migrations provide a structured, step-wise approach to database refactorings so it can evolve to meet changing requirements. Unlike code refactorings – which typically take a matter of minutes or seconds – database refactorings can take weeks or months to complete, with downtime for schema changes.

The bottom-line problem with the denormalized relational model is its resistance to the rapid evolution that today's business demands from applications. As we've seen in this data center example, the changes imposed on the initial whiteboard model from start to finish create a widening gulf between the conceptual world and the way the data is physically laid out.

This conceptual-relational dissonance prevents business and other non-technical stakeholders from further collaborating on the evolution of the system. As a result, the evolution of the application lags significantly behind the evolution of the business.

Now that we've thoroughly examined the relational data modeling process, let's turn to the graph data modeling approach.

Creating the Graph Data Model

As we've seen, relational data modeling divorces an application's storage model from the conceptual worldview of its stakeholders.

Relational databases – with their rigid schemas and complex modeling characteristics – are not an especially good tool for supporting rapid change. What we need is a model that is closely aligned with the domain, but that doesn't sacrifice performance, and that supports evolution while maintaining the integrity of the data as it undergoes rapid change and growth.

That model is the graph model. How, then, does the data modeling process differ? Let's begin.

In the early stages of graph modeling, the work is similar to the relational approach: Using lo-fi methods like whiteboard sketches, we describe and agree upon the initial domain. After that, our data modeling methodologies diverge.

Once again, here is our example data center domain modeled on the whiteboard:

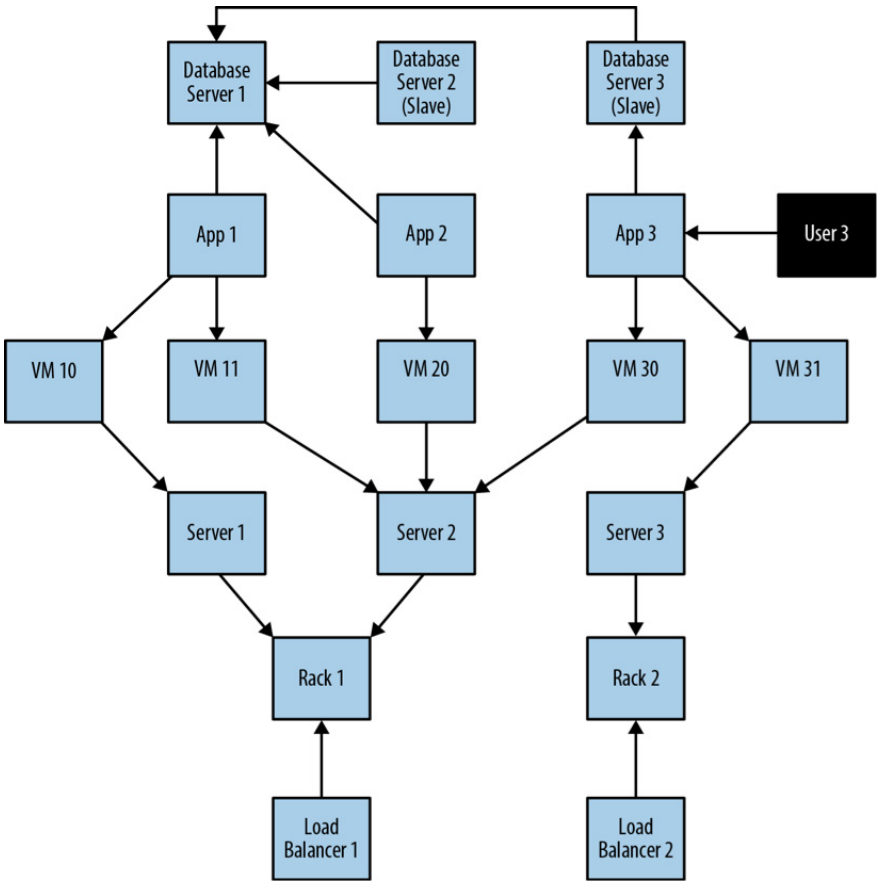

Figure 10: Our example data center domain with several application deployments.

Instead of turning our model into tables, our next step is to simply enrich our already graph-like structure. This enrichment aims to more accurately represent our application goals in the data model. In particular, we'll capture relevant roles as ***labels***, attributes as ***properties*** and connections to neighboring entities as ***relationships***.

By enriching this first-round domain model with additional properties and relationships, we produce a graph model attuned to our data needs; that is, we build our model to answer the kinds of questions our application will ask of its data.

To polish off our developing graph data model, we just need to ensure correct semantic context. We do this by creating named and directed relationships between nodes to capture the structural aspects of our domain.

Logically, that's ***all*** we need to do. No tables, no normalization, no denormalization. Once we have an accurate representation of our domain model, moving it into the database is trivial.

Instead of turning our model into tables, our next step is to simply enrich our already graph-like structure. No tables, no normalization, no denormalization. Once we have an accurate representation of our domain model, moving it into the database is trivial.

The Catch

So, what's the catch? With a graph database, what you sketch on the whiteboard is what you store in the database. It's that simple.

No catch.

After adding properties, labels and relationships, the resulting graph model for our data center scenario looks like this:

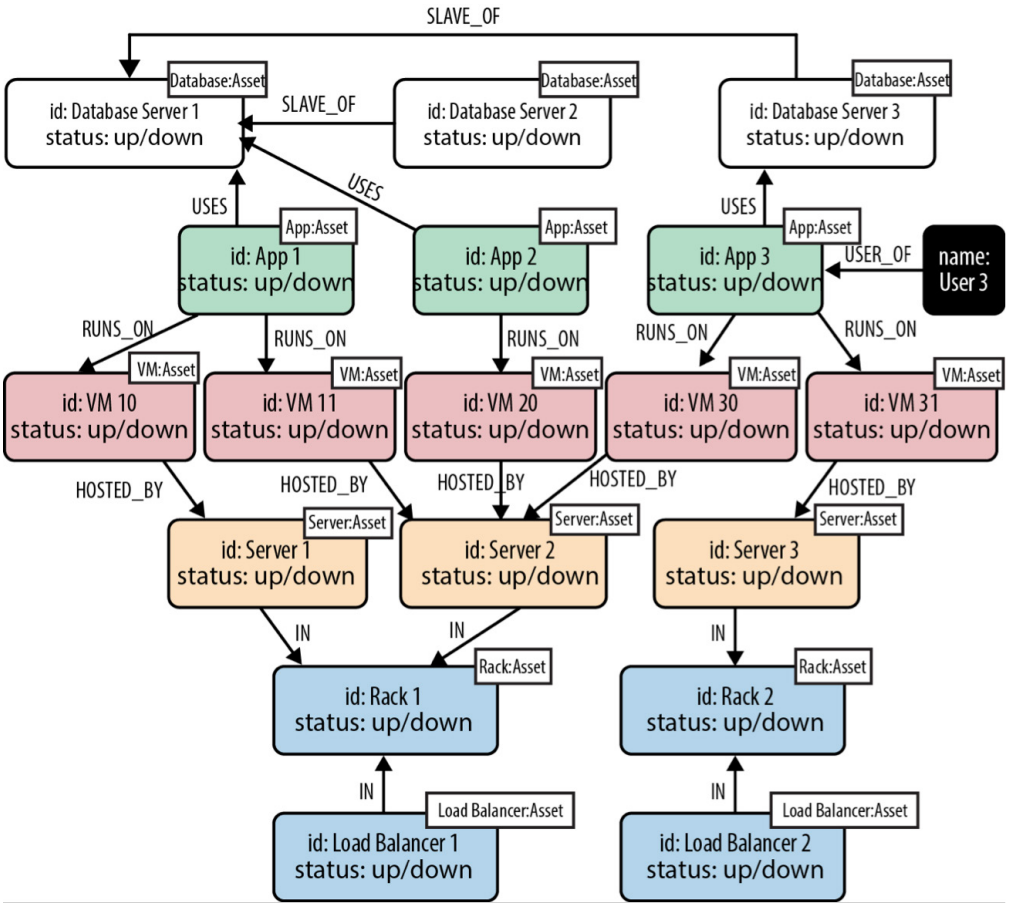

Figure 11: A full-fledged graph data model for our data center domain.

Note that most of the nodes here have two labels: both a specific type label (such as `Database`, `App` or `Server`), and a more general-purpose `Asset` label. This allows us to target particular types of assets with some of our queries, and all assets – irrespective of type – with other queries.

Compared to the finished relational database model (included again on the next page), ask yourself which of the two data models is easier to evolve, contains richer relationships and yet is still simple enough for business stakeholders to understand. We thought so.

With a graph database, what you sketch on the whiteboard is what you store in the database. It's that simple. No catch.

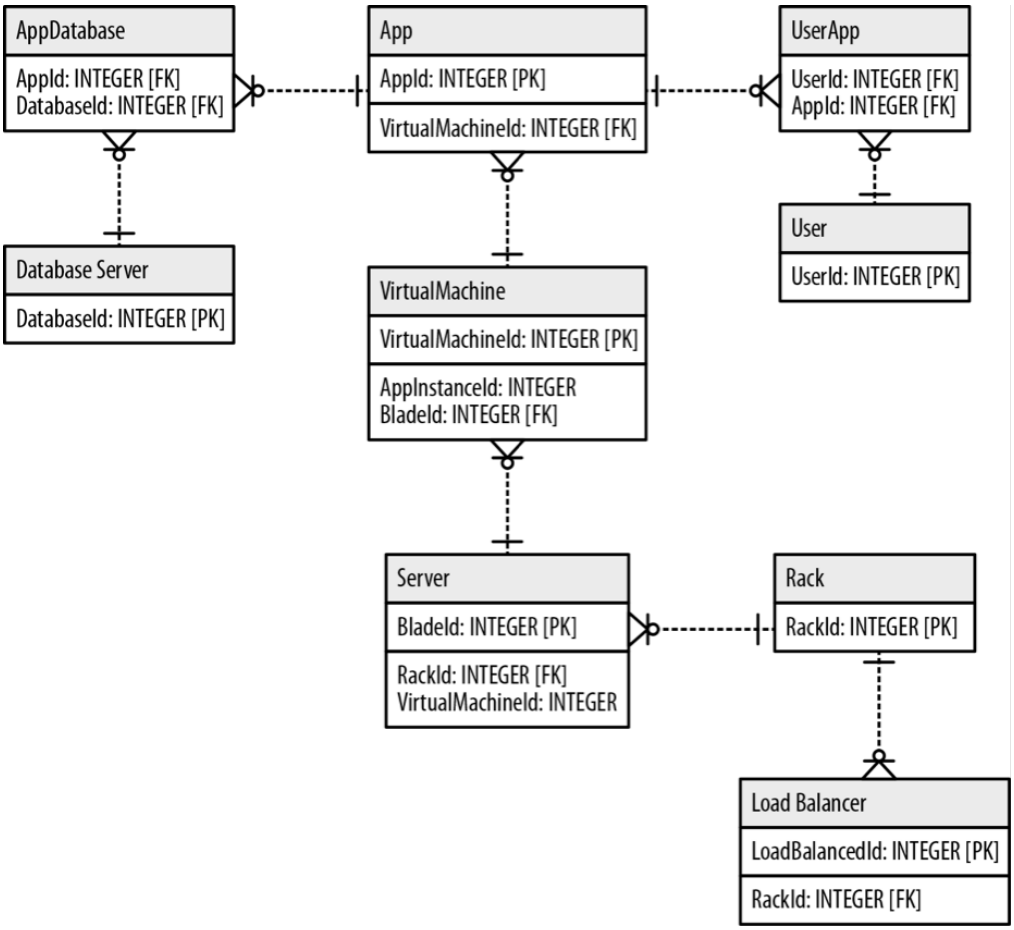

Figure 12: A full-fledged relational data model for our data center domain. This data model is significantly more complex – and less user friendly – than our graph model on the preceding page.

Conclusion

Don't forget: Data models are always changing. While it might be easy to dismiss an upfront modeling headache with the idea that you won't ever have to do it again, today's agile development practices will have you back at the whiteboard (or worse, calling a migration expert) sooner than you think.

Even so, data modeling is only a *part* of the database development lifecycle. Being able to query your data easily and efficiently – which often means real time – is just as important as having a rich and flexible data model. In the next chapter, we'll examine the differences between RDBMS and graph database query languages.

While it might be easy to dismiss an upfront modeling headache with the idea that you won't ever have to do it again, today's agile development practices will have you back at the whiteboard (or worse, calling a migration expert) sooner than you think.

Chapter 4:

Query Languages: SQL vs. Cypher

When it comes to a database query language, linguistic efficiency matters.

Querying relational databases is easy with SQL. As a declarative query language, SQL allows both for easy ad hoc querying in a database tool as well as specifying use-case related queries in your code. Even object-relational mappers use SQL under the hood to talk to the database.

But SQL runs up against major performance challenges when it tries to navigate connected data. For data-relationship questions, a single query in SQL can be many lines longer than the same query in a graph database query language like Cypher (more on Cypher below).

Lengthy SQL queries not only take more time to run, but they are also more likely to include human coding mistakes because of their complexity. In addition, shorter queries increase the ease of understanding and maintenance across your team of developers. For example, imagine if an outside developer had to pick through a complicated SQL query and try to figure out the intent of the original developer – trouble would certainly ensue.

But what level of efficiency gains are we talking about between SQL queries and graph queries? How much more efficient is one versus another? The answer: Fast enough to make a significant difference to your organization.

The efficiency of graph queries means they run in real time, and in an economy that runs at the speed of a single tweet, that's a bottom-line difference you can't afford to ignore.

The efficiency of graph queries means they run in real time, and in an economy that runs at the speed of a single tweet, that's a bottom-line difference you can't afford to ignore.

The Critical Relationship between Query Languages & Data Models

It's worth noting that a query language isn't just about asking (a.k.a. querying) the database for a particular set of results; it's also about modeling that data in the first place.

We know from the previous chapter that data modeling for a graph database is as easy as connecting circles and lines on a whiteboard. What you sketch on the whiteboard is what you store in the database.

On its own, this ease of modeling has many business benefits, the most obvious of which is that you can understand what your database developers are actually creating. But there's more to it: An intuitive model built with the right query language ensures there's no mismatch between how you ***built*** the data model and how you ***analyze*** it.

A query language represents its model closely. That's why SQL is all about tables and JOINs while Cypher is about relationships between entities. As much as the graph model is more natural to work with, so is Cypher as it borrows from the pictorial representation of circles connected with arrows which any stakeholder (whether technical or non-technical) can understand.

In a relational database, the data modeling process is so far abstracted from actual day-to-day SQL queries that there's a major disparity between analysis and implementation. In other words, the process of building a relational database model isn't fit for asking (and answering) questions efficiently from that same model.

Graph database models, on the other hand, not only communicate how your data is related, but they also help you clearly communicate the kinds of questions you want to ask of your data model. Graph models and graph queries are just two sides of the same coin.

The right database query language helps us traverse both sides.

An Introduction to Cypher, the Graph Query Language

Just like SQL is the standard query language for relational databases, Cypher is an open, multi-vendor query language for graph technologies. The advent of the openCypher project has expanded the reach of Cypher well beyond just Neo4j, its original sponsor.

Cypher – also a declarative query language – is built on the basic concepts and clauses of SQL but with added graph-specific functionality, making it simple to work with a rich graph model without being overly verbose.

(Note: This introduction isn't a reference document for Cypher but merely a high-level overview.)

Cypher is designed to be easily read and understood by developers, database professionals and business stakeholders alike. It's easy to use because it matches the way we intuitively describe graphs using diagrams.

If you have ever tried to write a SQL statement with a large number of JOINs, you know that you quickly lose sight of what the query actually does, due to all the technical noise. In contrast, Cypher syntax stays clean and focused on domain concepts since queries are expressed visually.

The basic notion of Cypher is that it allows you to ask the database to find data that matches a specific pattern. Colloquially, we might ask the database to "find things like this," and the way we describe what "things like this" look like is to draw them using ASCII art.

Consider the social graph below describing three mutual friends:

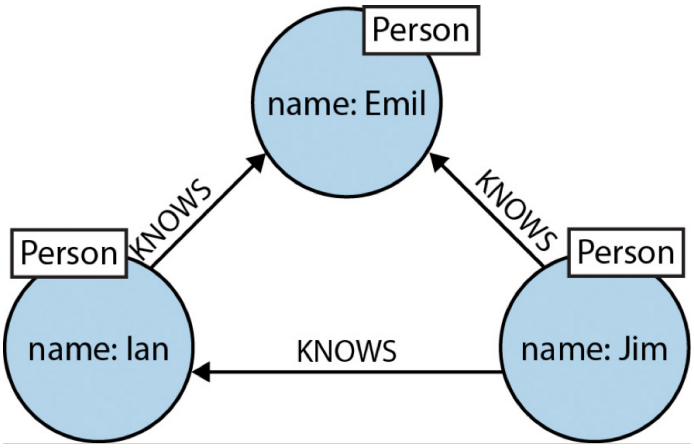

Figure 13: A social graph describing the relationship between three friends.

If we want to express the pattern of this basic graph in Cypher, we would write:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

This Cypher pattern describes a path which forms a triangle that connects a node we call `jim` to the two nodes we call `ian` and `emil`, and which also connects the `ian` node to the `emil` node. As you can see, Cypher naturally follows the way we draw graphs on the whiteboard.

Now, while this Cypher pattern describes a simple graph structure it doesn't yet refer to any particular data in a graph database. To bind the pattern to specific nodes and relationships in an existing dataset, we first need to specify some property values and node labels that help locate the relevant elements in the dataset.

```
(emil:Person {name:'Emil'})
     <-[:KNOWS]-(jim:Person {name:'Jim'})
     -[:KNOWS]->(ian:Person {name:'Ian'})
     -[:KNOWS]->(emil)
```

Here's our more fleshed-out Cypher statement:

Here we've bound each node to its identifier using its `name` property and `Person` label. The `emil` identifier, for example, is bound to a node in the dataset with a label `Person` and a `name` property whose value is `emil`. Anchoring parts of the pattern to real data in this way is normal Cypher practice.

Using ASCII characters to represent nodes and relationships, we draw the data we're interested in.

The RDBMS Developer's Guide to Cypher Clauses

Like most query languages, Cypher is composed of clauses.

The simplest queries consist of a `MATCH` clause followed by a `RETURN` clause. Here's an example of a Cypher query that uses these two clauses to find the mutual friends of a user named `jim` (from our social graph pictured on the previous page):

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b:Person)-
      [:KNOWS]->(c:Person), (a)-[:KNOWS]->(c)
RETURN b, c
```

Let's look at each clause in further detail:

MATCH

The `MATCH` clause is at the heart of most Cypher queries.

Using ASCII characters to represent nodes and relationships, we draw the data we're interested in. We draw nodes with parentheses, like in the example from the above query:

```
(a:Person {name:'Jim'})
(b:Person)
(c:Person)
(a)
```

We draw relationships using using pairs of dashes with greater-than or less-than signs (`-->` and `<--`) where the < and > signs indicate relationship direction. Between the dashes, relationship names are enclosed by square brackets and prefixed by a colon, like in this example from the query above:

```
-[:KNOWS]->
```

Node labels are also prefixed by a colon. As you see in the first node of the query, `Person` is the applicable label:

```
(a:Person … )
```

Cypher matches the remainder of the pattern to the graph immediately surrounding this anchor point based on the provided information on relationships and neighboring nodes.

Node (and relationship) property key-value pairs are then specified within curly braces, like in this example:

```
( … {name:'Jim'})
```

In our original example query, we're looking for a node labeled `Person` with a `name` property whose value is `Jim`. The return value from this lookup is bound to the identifier `a`. This identifier allows us to refer to the node that represents `Jim` throughout the rest of the query.

It's worth noting that this pattern

```
(a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
```

could, in theory, occur many times throughout our graph, especially in a large user dataset.

To confine the query, we need to anchor some part of it to one or more places in the graph. In specifying that we're looking for a node labeled `Person` whose name property value is `Jim`, we've bound the pattern to a specific node in the graph — the node representing Jim.

Cypher then matches the remainder of the pattern to the graph immediately surrounding this anchor point based on the provided information on relationships and neighboring nodes. As it does so, it discovers nodes to bind to the other identifiers. While `a` will always be anchored to `Jim`, `b` and `c` will be bound to a sequence of nodes as the query executes.

RETURN

This clause specifies which expressions, relationships and properties in the matched data should be returned to the client. In our example query, we're interested in returning the nodes bound to the b and c identifiers.

Other Cypher Clauses

Other clauses you can use in a Cypher query include:

WHERE

Provides criteria for filtering pattern matching results.

CREATE and CREATE UNIQUE

Creates nodes and relationships.

MERGE

Ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new nodes and relationships.

DELETE/REMOVE

Removes nodes, relationships and properties.

SET

Sets property values and labels.

ORDER BY

Sorts results as part of a `RETURN`.

SKIP LIMIT

Skips results at the top and limits the number of results.

FOREACH

Performs an updating action for each element in a list.

UNION

Merges results from two or more queries.

WITH

Chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix.

If these clauses look familiar – especially if you're an RDBMS developer – that's great! Cypher is intended to be easy to learn for SQL veterans while also being simple enough for beginners. (Click here for the most up-to-date Cypher Refcard to take a deeper dive into the Cypher query language.)

SQL vs. Cypher Query Examples: The Good, the Bad and the Ugly

Now that you have a basic understanding of Cypher, it's time to compare it side by side with SQL to realize the linguistic efficiency of the former – and the inefficiency of the latter – when it comes to queries around connected data.

Our first example uses the organizational domain (from Chapter 3) pictured below as a relational data model:

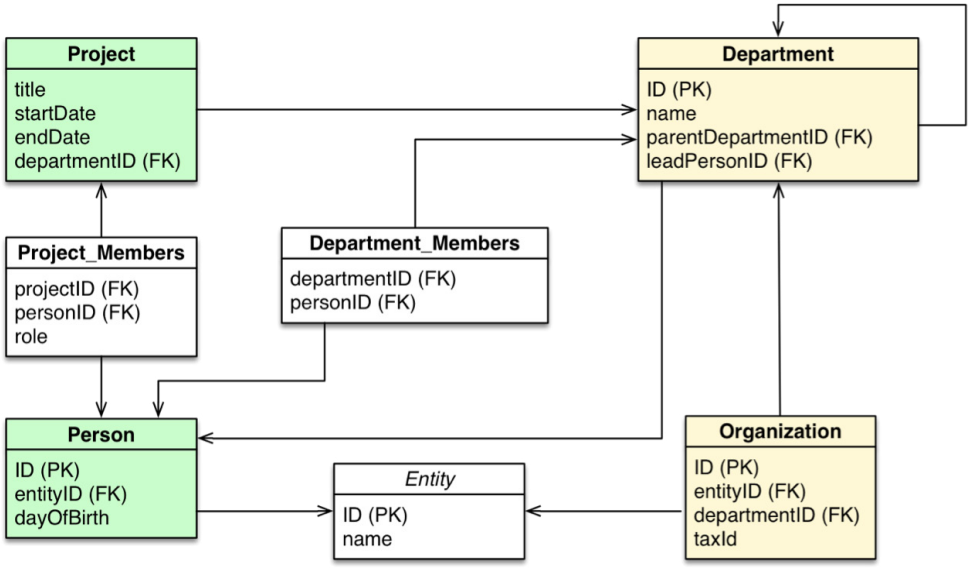

Figure 14: A relational data model of an organizational domain.

In the organizational domain depicted in the model above, what would a SQL statement that lists the *employees in the "IT Department"* look like? And how does that statement compare to a Cypher statement?

SQL Statement:

```
SELECT name FROM Person

LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId

LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId

WHERE Department.name = "IT Department"
```

Cypher Statement:

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)

WHERE d.name = "IT Department"

RETURN p.name
```

In this example on the previous page, the Cypher query is half the length of the SQL statement and significantly simpler. Not only would this Cypher query be faster to create and run, but it also reduces chances for error.

Now for another example, this one more extreme. We'll start with the Cypher query:

Cypher Statement:

```
MATCH (u:Customer {customer_id:'customer-one'})-
     [:BOUGHT]->(p:Product)<- [:BOUGHT]-
     (peer:Customer)-[:BOUGHT]->(reco:Product)

WHERE not (u)-[:BOUGHT]->(reco)

RETURN reco as Recommendation, count(*) as Frequency

ORDER BY Frequency DESC LIMIT 5;
```

This Cypher query says that for each customer who bought a product, look at the products that peer customers have purchased and add them as recommendations. The `WHERE` clause removes products that the customer has already purchased, since we don't want to recommend something the customer already bought.

Each of the arrows in the `MATCH` clause of the Cypher query represents a relationship that would be modeled as a many-to-many JOIN table in a relational model with two JOINs each. So even this simple query encompasses potentially six JOINs across tables. Here's the equivalent query in SQL:

SQL Statement:

```
SELECT product.product_name as Recommendation, count(1) as Frequency
FROM product, customer_product_mapping, (SELECT cpm3.product_id,
cpm3.customer_id
      FROM Customer_product_mapping cpm, Customer_product_mapping cpm2,
      Customer_product_mapping cpm3
      WHERE cpm.customer_id = 'customer-one'
      and cpm.product_id = cpm2.product_id
      and cpm2.customer_id != 'customer-one'
      and cpm3.customer_id = cpm2.customer_id
      and cpm3.product_id not in (select distinct product_id
      FROM Customer_product_mapping cpm
      WHERE cpm.customer_id = 'customer-one')
      ) recommended_products
WHERE customer_product_mapping.product_id = product.product_id
and customer_product_mapping.product_id in recommended_products.product_id
and customer_product_mapping.customer_id = recommended_products.customer_id
GROUP BY product.product_name
ORDER BY Frequency desc
```

This SQL statement is ***three times*** as long as the equivalent Cypher query. It will not only suffer from performance issues due to the JOIN complexity but will also degrade in performance as the dataset grows.

Conclusion

If application performance is a priority, then your database query language matters.

SQL is well-optimized for relational database models, but once it has to handle complex, relationship-oriented queries, its performance quickly degrades. In these instances, the root problem doesn't lie with SQL but with the relational model itself, which isn't designed to handle connected data.

For domains with highly connected data, the graph model is a must, and as a result, so is a graph query language like Cypher. If your development team comes from an SQL background, then Cypher will be easy to learn and even easier to execute.

When it comes to your next graph-powered, enterprise-level application, you'll be glad that the query language underpinning it all is built for speed and efficiency.

For domains with highly connected data, the graph model is a must, and as a result, so is a graph query language like Cypher.

Chapter 5:

Deployment Paradigms: Bringing Graphs into Your Architecture

Whether you're ready to move your entire legacy RDBMS into a graph database, you're syncing databases for polyglot persistence or you're just conducting a brief proof of concept, at some point you'll want to bring a graph database into your organization or architecture.

Once you've decided on your deployment paradigm, you'll then need to move some (or all) of your data from your relational database into a graph database. In this chapter, we'll show you how to make that process as smooth and seamless as possible.

Your first step is to ensure you have a proper understanding of the native graph property model (i.e., nodes, relationships, labels, properties and relationship-types), particularly as it applies to your given domain.

In fact, you should at least complete a basic graph model on a whiteboard before you begin your data import. Knowing your data model ahead of time – and the deployment paradigm in which you'll use it – makes the import process significantly less painful.

There are three main paradigms to deploying a graph database relative to your RDBMS. Which paradigm is best for your application or architecture depends on your particular goals.

Three Deployment Paradigms for Graphs and RDBMS

There are three main paradigms to deploying a graph database relative to your RDBMS. Which paradigm is best for your application or architecture depends on your particular goals.

Below, you can see each of the paradigms for deploying both a relational and graph database:

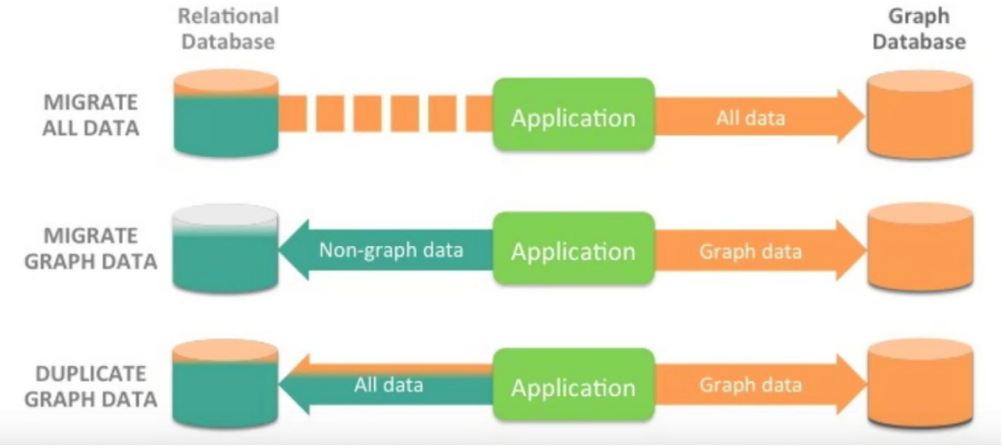

Figure 15: The three most common paradigms for deploying relational and graph databases.

First, some development teams decide to abandon their relational database altogether and migrate all of their data into a graph database. This is typically a one-time, bulk migration.

Second, other developers continue to use their relational database for any use case that relies on non-graph, tabular data. Then, for any use cases that involve a lot of JOINs or data relationships, they store that data in a graph database.

Third, some development teams duplicate ***all*** of their data into both a relational database and a graph database. That way, data can be queried in whatever form is the most optimal for the queries they're trying to run.

The second and third paradigms are considered polyglot persistence, since both approaches use a data store according to its strengths. While this introduces additional complexity into an application's architecture, it often results in getting the most optimized results from the best database for the query.

None of these is the "correct" paradigm for deploying an RDBMS and a graph. Your team should consider your application goals, frequent use cases and most common queries and choose the appropriate solution for your particular environment.

Extracting Your Data from an RDBMS

No matter your given paradigm, if you decide you need to import your relational data into a graph database, the first step is to extract (or copy) it from your existing RDBMS.

Most all relational databases allow you to dump both whole tables or whole datasets, as well as carry results to CSV and to post queries. These tasks are usually just a copy function of the database itself. Of course, in many cases the CSV file resides on the database, so you have to download it from there, which can be a challenge.

Another option is to access your relational database with a database driver like JDBC or another driver to extract the datasets you want to pull out.

Also, if you want to set up a syncing mechanism between your relational and graph databases, then it makes sense to regularly pull the given data according to a timestamp or another updated flag so that data is synced into your graph.

Another facet to consider is that many relational databases aren't designed or optimized for exporting large amounts of data within a short time period. So if you're trying to migrate data directly from an RDBMS to a graph, the process might stall significantly.

For example, in one case a Neo4j customer had a large social network stored in a MySQL cluster. Exporting the data from the MySQL database took three days; importing it into Neo4j took just three hours.

In one case, exporting the data from the MySQL database took three days; importing it into Neo4j took just three hours.

One final tip before you begin: When you write to disk, be sure to disable virus scanners and check your disk schedule so you get the highest disk performance possible. It's also worth checking any other options that might increase performance during the import process.

Importing Data via `LOAD CSV`

The easiest way to import data from your relational database is to create a CSV dump of individual entity-tables and JOIN-tables. The CSV format is the lowest common denominator of data formats between a variety of different applications. While the CSV format itself is unpopular, it's also the easiest to work with when it comes to importing data into a graph database.

In Neo4j specifically, `LOAD CSV` is a Cypher keyword that allows you to load CSV files from HTTP or file URLs into your database. Each row of data is made available to your Cypher statement and then from those rows, you can actually create or update nodes and relationships within your graph.

The `LOAD CSV` command is a powerful way of converting flat data (i.e., CSV files) into connected graph data. `LOAD CSV` works both with single-table CSV files as well as with files that contain a fully denormalized table or a JOIN of several tables.

`LOAD CSV` allows you to convert, filter or de-structure import data during the import process. You can also use this command to split areas, pull out a single value or iterate over a certain list of attributes and then filter them out as attributes.

Finally, with `LOAD CSV` you can control the size of transactions so you don't run into memory issues with a certain keyword, and you can run `LOAD CSV` via the Neo4j shell (and not just the Neo4j browser), which makes it easier it script your data imports.

In summary, you can use Cypher's `LOAD CSV` command to:

- Ingest data, accessing columns by header name or offset
- Convert values from strings to different formats and structures (`toFloat`, `split`, ...)
- Skip rows to be ignored
- `MATCH` existing nodes based on attribute lookups
- `CREATE` or `MERGE` nodes and relationships with labels and attributes from the row data
- `SET` new labels and properties or `REMOVE` outdated ones

A `LOAD CSV` Example

Here's a brief example of importing a CSV file into Neo4j using the `LOAD CSV` Cypher command.

Example file: `persons.csv`

```
name;email;dept
"Lars Higgs";"lars@higgs.com";"IT-Department"
"Maura Wilson";"maura@wilson.com";"Procurement"
```

Cypher Statement:

```
LOAD CSV FROM 'file:///data/persons.csv' WITH HEADERS AS line
FIELDTERMINATOR ";"
MERGE (person:Person {email: line.email}) ON CREATE SET p.name = line.name
MATCH (dep:Department {name:line.dept})
CREATE (person)-[:EMPLOYEE]->(dept)
```

You can import multiple CSV files from one or more data sources (including your RDBMS) to enrich your core domain model with other information that might add interesting insights and capabilities.

Other, dedicated import tools help you import larger volumes (10M+ rows) of data efficiently, as described below.

The Command-Line Bulk Loader

The *neo4j-import* command is a scalable input tool for bulk inserts. This tool takes CSV files and scales them across all of your available CPUs and disk capacity, putting the data into a stage architecture where each input step is parallelized if possible.

Then, the tool stages step-by-step input using some advanced in-memory compression for creating new graph structures.

The command-line bulk loader is lightning fast, able to import up to one million records per second and handle large datasets of several billion nodes, relationships and properties. Note that because of these performance optimizations the ***neo4j-import*** tool can only be used for initial database population.

Loading Data Using Cypher

For importing data, you can also use the Neo4j REST API to run Cypher statements yourself. With this API, you can run, create, update and merge statements using Cypher.

The transactional Cypher HTTP endpoint is available to all drivers. You can also use the HTTP endpoint directly from an HTTP client or an HTTP library in your language.

In one test, one million nodes and relationships per second were inserted with highly concurrent Cypher statements using this method.

Using the HTTP endpoint (or another API), you can pull the data out of your relational database (or other data source) and convert it into parameters for Cypher statements. Then you can batch and control import transactions from there.

From Neo4j 2.2 onwards, Cypher also works really well with highly concurrent writes. In one test, one million nodes and relationships per second were inserted with highly concurrent Cypher statements using this method.

The Cypher-based loading method works with a number of different drivers, including the JDBC driver. If you have an ETL tool or Java program that already uses a JDBC tool, you can use Neo4j's JDBC driver to import data into Neo4j because Cypher statements are just query strings (more on the JDBC driver in Chapter 6). In this scenario, you can provide parameters to your Cypher statements as well.

Other RDBMS-to-Graph Import Resources

This chapter has only covered the three most common methods for importing data into a graph database from a relational store. The following are further resources on additional methods for data import, as well as more in-depth guides on the three methods discussed above:

- Guide: Data Import
- Manual: LOAD CSV
- Webinar: Data Import
- Guide: CSV Import
- Tool: Direct RDBMS Import
- Tool: SQL to Neo4j Import
- Blog: Importing AdventureWorks Data into Neo4j
- Neo4j for Relational MetaData (SQLServer)

Chapter 6:

Drivers: Connecting to a Graph Database

Up to this point in the book, we've covered graph databases in general, only using Neo4j when examples were required. However, when it comes to connecting your application to a graph database, specifics are essential.

In this chapter, we'll discuss the language drivers and APIs specific to Neo4j with plenty of resources for further exploration. At this point, if you are curious about other, non-Neo4j graph databases, we encourage you to explore the available drivers within their respective communities.

The Neo4j Browser

If you've installed and started Neo4j as a server on your system, the good news is that you can already interact with the database via the built-in Neo4j Browser application.

The Neo4j Browser is similar to (but more feature-rich than) SQL*Plus, giving you a familiar and comfortable way to interact directly with your new graph database. The easiest way to connect to the Neo4j Browser is through the `localhost:7474` port.

Read this article for more information on the Neo4j Browser.

However, even with the Neo4j Browser, you'll still want your application to connect to Neo4j through other means – most often through a driver for your programming language of choice. In this chapter, we'll cover the various ways to connect with Neo4j, including the HTTP-API and various language drivers.

The Neo4j Browser is similar to (but more feature-rich than) SQL*Plus, giving you a familiar and comfortable way to interact directly with your new graph database.

Connecting to Neo4j via REST API

If you want to access Neo4j programmatically, you can do so with the REST API, which allows you to:

- `POST` one or more Cypher statements with parameters per request to the server
- Keep transactions open over multiple requests
- Choose different result formats
- Execute management options or introspect the database

This API can then be used directly via an HTTP library or through a language driver (more on those far below).

Let's look at one of the underlying remote API endpoints that Neo4j offers to execute queries. A simple HTTP Cypher request – executable in the Neo4j Browser – would look like this:

```
:POST /db/data/transaction/commit {"statements":[
      {"statement":"CREATE (p:Person {firstName:{name}})
RETURN p",
       "parameters":{"name":"Daniel"}}
    ]}
```

Another example: An HTTP request that executes Cypher to create a Person would look like this, here shown with the plain JSON response:

```
:POST http://localhost:7474/db/data/transaction/commit
 {"statements":[
   {"statement":"CREATE (p:Person {name:{name}}) RETURN p",
"parameters":{"name":"Daniel"}}
  ]}
->
{"results":[{"columns":["p"],"data":[{"row":[{"name":"Daniel"}]}]}],
,"errors":[]}
```

The language drivers discussed below use the same APIs under the hood, but make them available in a convenient way.

Connecting to Neo4j via Language Drivers

In most cases, you won't want to connect to Neo4j manually, but with a driver or connector library designed for your stack or programming language.

Thanks to the Neo4j community, there are Neo4j drivers for almost every popular programming language, most of which mimic existing database driver idioms and approaches.

Some of the most popular language drivers for Neo4j include:

- Java
- .NET
- JavaScript
- Python
- Ruby
- PHP
- R
- Go
- Clojure
- Perl
- Haskell

Below, we'll take a closer look at two drivers with the greatest familiarity for RDBMS developers: JDBC and Spring Data.

Using Neo4j Server with JDBC

If you're a Java developer, you're probably familiar with Java Database Connectivity (JDBC) as a way to work with relational databases, whether directly or via abstractions like Spring's `JDBCTemplate` or MyBatis. Many other tools use JDBC drivers to interact with relational databases for business intelligence, data management or ETL (Extract, Transform, Load).

Cypher – like SQL – is a textual and parameterizable query language capable of returning tabular results, so Neo4j supports the JDBC APIs through a Neo4j-JDBC driver.

Let's look at an example query using the JDBC driver. Working from the organizational data model example in Chapter 3, here's what a query for finding John's department would look like using the Neo4j-JDBC driver:

```
Connection con =
DriverManager.getConnection("jdbc:neo4j://
     localhost:7474/");

String query =
    "MATCH (:Person {name:{1}})-[:EMPLOYEE]-
           (d:Department) RETURN d.name as dept";
try (PreparedStatement stmt = con.prepareStatement
     (QUERY)) {
     stmt.setString(1,"John");
     ResultSet rs = stmt.executeQuery();
     while(rs.next()) {
          String department = rs.getString("dept");
          ....
     }
}
```

Spring Data Neo4j integrates the Neo4j-OGM library to provide fast and comprehensive object graph mapping.

More details on how to use the Neo4j-JDBC driver can be found with our example project for Java-JDBC. There, we implement the backend as a minimal Java web app that leverages Neo4j-JDBC to connect to Neo4j Server.

Using Spring Data Neo4j

Spring Data Neo4j integrates the Neo4j-OGM library to provide fast and comprehensive object graph mapping. Additionally, it provides support for Spring conversions, transaction handling, Spring-Data-Repositories, Spring-Data REST and Spring-Boot.

To get started, add Spring Data Neo4j as a dependency, then configure the necessary beans in your Java `config`. Then you can use `Neo4jTemplate` to manage your entities and define Spring-Data-Repositories as convenient interfaces to your persistence layer.

Here are more resources on using Spring Data Neo4j:

- Quickstart guide to using Spring Data Neo4j 4 (SDN4)
- Spring Data Neo4j 4 Example Projects
- Spring Data Neo4j on GitHub
- Reference Documentation, JavaDoc and ChangeLog
- Article: Introducing SDN4 by Luanne Misquitta
- Video: Introducing SDN4

This chapter gives only a glimpse at the many ways to connect your application with Neo4j. Whether you're connecting via the Neo4j Browser, via the REST API or via a language driver of your choice, you can find more in-depth and up-to-date resources in the Developer Guides for Neo4j.

Conclusion:

Never Write Another JOIN

The recent proliferation of NoSQL database technologies is a testament to the fact that relational databases aren't the right tool for every job.

While relational databases are powerful tools for the right use case and the right architecture, today's user and business needs are changing. As we've seen above, today's complex, real-world data is increasing in volume, velocity and variety, and the data relationships – which are often more valuable than the data itself – are growing at an even faster rate.

Leveraging those connected data relationships are where graph databases shine. When you make the switch to graph databases – whether as your main or secondary data store – you can capture that rich relationship information in a way your RDBMS never can.

Furthermore, using a graph database for your next application allows you to match changes in your database schema with the speed of business agility. And once your application is in production, a graph database allows you to query connected data at speeds and performance levels that no RDBMS can match.

Because of the natural fit of the graph database model to business domains and processes, Forrester Research estimates that at least 25% of enterprises worldwide will use graph databases by 2017, while Gartner Research reports that graph databases are the fastest-growing category in database management systems, and that "by the end of 2018, 70 percent of leading organizations will have one or more pilot or proof-of-concept efforts underway utilizing graph databases."

Instead of imposing a connected data model onto a traditional RDBMS – with the accompanying struggle against intensive and frustrating JOINs – it's time to use a graph database for every use case where data relationships matter most.

In the end, business won't wait and neither will users. When you choose the right tool for your connected-data application, you never have to write another JOIN. Even better: You reduce query times from minutes to milliseconds.

With so many ways to get started quickly, mastering graph database development is one of the best time investments you can make.

Gartner Research reports by the end of 2018, 70 percent of leading organizations will have one or more pilot or proof-of-concept efforts underway utilizing graph databases.

This ebook only scratches the surface when it comes to how today's RDBMS developer can best take advantage of graph databases.

Other Resources:

More on Graph Databases for the RDBMS Developer

This ebook only scratches the surface when it comes to how today's RDBMS developer can best take advantage of graph databases.

For more insights, tips and tricks on the intersection of relational and graph databases, explore any of the many resources below:

Articles and Presentations:

- The Graph Databases for Beginners blog series
- Developer Guide: From Relational to Neo4j
- Presentation: From Relational to (Big) Graph
- White Paper: Overcoming SQL Strain and SQL Pain

Webinars:

- Webinar: From RDBMS to Graphs
- Webinar: Relational to Graph: Data Modeling
- Webinar: Relational to Graph: Importing Data into Neo4j

Books:

- O'Reilly's ***Graph Databases*** ebook
- ***Learning Neo4j*** ebook

Trainings:

- Online Training: Getting Started with Neo4j
- Classroom Trainings

Neo Technology exists to help the world make sense of data. That's why Neo Technology created Neo4j, the world's leading graph database. Organizations worldwide use Neo4j to extract real-time insights not only from their data but also data relationships. Neo Technology has brought the power of graph databases to organizations both large and small – from enterprises like Walmart, eBay, UBS, Cisco, HP, Telenor and Lufthansa to young startups like Polyvore, Medium and Zephyr Health. These businesses have used Neo4j to build solutions as varied as personalized recommendation engines, identity and access management models, social networks, network monitoring tools and master data management applications. From its inception, Neo Technology has always valued relationships, and now organizations worldwide use the Neo4j graph database to unlock the business value from their data relationships.

For more information on Neo4j, contact us via email or phone:
1-855-636-4532
info@neotechnology.com